LogiCORE™ IP
SPI-4.2 Core v8.5

Getting Started Guide

UG154 March 24, 2008

"Xilinx" and the Xilinx logo shown above are registered trademarks of Xilinx, Inc. Any rights not expressly granted herein are reserved.

CoolRunner, RocketChips, Rocket IP, Spartan, StateBENCH, StateCAD, Virtex, XACT, XC2064, XC3090, XC4005, and XC5210 are registered trademarks of Xilinx, Inc.

The shadow X shown above is a trademark of Xilinx, Inc.

ACE Controller, ACE Flash, A.K.A. Speed, Alliance Series, AllianceCORE, Bencher, ChipScope, Configurable Logic Cell, CORE Generator, CoreLINX, Dual Block, EZTag, Fast CLK, Fast CONNECT, Fast FLASH, FastMap, Fast Zero Power, Foundation, Gigabit Speeds...and Beyond!, HardWire, HDL Bencher, IRL, J Drive, JBits, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroBlaze, MicroVia, MultiLINX, NanoBlaze, PicoBlaze, PLUSASM, PowerGuide, PowerMaze, QPro, Real-PCI, RocketIO, SelectIO, SelectRAM, SelectRAM+, Silicon Xpresso, Smartguide, Smart-IP, SmartSearch, SMARTswitch, System ACE, Testbench In A Minute, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex-II Pro, Virtex-II EasyPath, Virtex-4, Wave Table, WebFITTER, WebPACK, WebPOWERED, XABEL, XACT-Floorplanner, XACT-Performance, XACTstep Advanced, XACTstep Foundry, XAM, XAPP, X-BLOX +, XC designated products, XChecker, XDM, XEPLD, Xilinx Foundation Series, Xilinx XDTV, Xinfo, XSI, XtremeDSP and ZERO+ are trademarks of Xilinx, Inc.

The Programmable Logic Company is a service mark of Xilinx, Inc.

All other trademarks are the property of their respective owners.

Xilinx, Inc. does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx, Inc. reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx, Inc. will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx provides any design, code, or information shown or described herein "as is." By providing the design, code, or information as one possible implementation of a feature, application, or standard, Xilinx makes no representation that such implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Xilinx expressly disclaims any warranty whatsoever with respect to the adequacy of any such implementation, including but not limited to any warranties or representations that the implementation is free from claims of infringement, as well as any implied warranties of merchantability or fitness for a particular purpose. Xilinx, Inc. devices and products are protected under U.S. Patents. Other U.S. and foreign patents pending. Xilinx, Inc. does not represent that devices shown or products described herein are free from patent infringement or from any other third party right. Xilinx, Inc. assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx, Inc. will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.

The contents of this manual are owned and copyrighted by Xilinx. Copyright 2004-2008 Xilinx, Inc. All Rights Reserved. Except as stated herein, none of the material may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Any unauthorized use of any material contained in this manual may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/30/04	1.0	Initial Xilinx release.
11/11/04	1.1	Document updated to support SPI-4.2 core v7.1.
04/28/05	1.2	Document updated to support SPI-4.2 core v7.2 and Xilinx ISE v7.1i.
08/31/05	2.0	Updated ISE service pack information.
1/18/06	3.0	Updated ISE to v8.1i, release date
7/13/06	4.0	Added support for Virtex-5, ISE to v8.2i, advanced version number and release date.
9/21/06	4.1	Updted for IP2i minor release. Removed Simulating the Dynamic Alignment Sink core section from the example design chapter.
2/15/07	4.2	Updated system requirements, ISE version, and applied new directory structure template to Chapter 4.
8/08/07	4.3	Updated for IP1 Jade Minor release. ISE version to 9.2i.
3/24/08	4.4	Updated core to v8.5, updated supported tool versions, and release date.

Table of Contents

Schedule of Figures 7

Schedule of Tables 9

Preface: About This Guide

Contents 11

Conventions 11

Typographical 12

Online Document 12

Chapter 1: Introduction

System Requirements 13

About the Core 13

Recommended Design Experience 13

Additional Core Resources 14

Technical Support 14

Feedback 14

Core 14

Document 14

Chapter 2: Licensing the Core

Before you Begin 15

License Options 15

Simulation-Only Evaluation 15

Full System Hardware Evaluation 15

Full 16

Obtaining Your License 16

Installing Your License File 17

Chapter 3: Quick Start Example Design

Overview 19

Generating the Core 19

Implementing the Example Design 21

Running the Simulation 21

Setting up for Simulation 21

Functional Simulation 21

Timing Simulation 22

Chapter 4: Detailed Example Design

Directory and File Contents 26

<project directory> 26

<project directory>/<component name> 26
<component name>/doc 26
<component name>/example design 27
<component name>/implement 28
implement/results 29
<component name>/simulation 29
simulation/functional 30
simulation/timing 31
Implementation and Simulation Scripts 31
Simulation Script Details 32
Example Design Configuration 32
Loopback Module 33
Basic Loopback Operation 33
Demonstration Test Bench 34
Clock Generator 35
Startup Module 36
Stimulus Module 37
Procedures Module 38
Data Monitor 38
Status Monitor 38
Customizing the Demonstration Test Bench 39
Test Case Package 39
Testcase Module 41
Calendar Sequence Files (Sink and Source) 43

Appendix A: VHDL Details

Procedures Module 45

Appendix B: Verilog Details

Procedures Module 49
Random Testcase Sample Code 51

Appendix C: Data and Status Monitor Warnings

Appendix D: Timing Simulation Warning and Error Messages

Schedule of Figures

Chapter 3: Quick Start Example Design

Figure 3-1: **Core Customization GUI Main Window** 20

Chapter 4: Detailed Example Design

Figure 4-1: **Example Design Configuration** 33
Figure 4-2: **Demonstration Test Bench Connections** 34
Figure 4-3: **Test Bench Modules** 35
Figure 4-4: **Startup State Diagram** 36

Schedule of Tables

Chapter 4: Detailed Example Design

Table 4-1: **Project Directory** 26
Table 4-2: **Component Name Directory** 26
Table 4-3: **Doc Directory** 26
Table 4-4: **Example Design Directory** 27
Table 4-5: **Implement Directory** 28
Table 4-6: **Results Directory** 29
Table 4-7: **Simulation Directory** 29
Table 4-8: **Functional Directory** 30
Table 4-9: **Timing Directory** 31
Table 4-10: **Testcase Package User-Defined Constants** 39
Table 4-11: **Useful Testcase Signals** 42
Table 4-12: **Testcase Module Request Signals** 42

Appendix A: VHDL Details

Table A-1: **send_packet (PBr, addr, bytes) Inputs** 45
Table A-2: **send_user_data (PBr, SOP, EOP, Err, Addr, bytes) Inputs** 46
Table A-3: **send_idles (PBr, cycles) Inputs** 46
Table A-4: **send_training (PBr, patterns) Inputs** 46
Table A-5: **sop_spacing (PBr, Bytes1, Err1, Addr1, EOP2, Err2, Addr2, Bytes2, num_cycles) Inputs** 46
Table A-6: **send_status (PBt, channel, value) Inputs** 47
Table A-7: **get_status (PBt, channel) Inputs** 47

Appendix B: Verilog Details

Table B-1: **send_packet (Addr, bytes) Inputs** 49
Table B-2: **send_user_data (SOP, EOP, Err, Addr, bytes) Inputs** 50
Table B-3: **send_idles (cycles) Inputs** 50
Table B-4: **send_training (patterns) Inputs** 50
Table B-5: **sop_spacing (Bytes1, Err1, Addr1, EOP2, Err2, Addr2, Bytes2, num_cycles) Inputs** 50
Table B-6: **send_status (channel, value) Inputs** 51
Table B-7: **get_status (channel) Inputs** 51

About This Guide

This guide provides information about generating the Xilinx LogiCORE™ IP SPI-4.2 core, customizing and simulating the core using the provided example design, and running the design files through implementation using the Xilinx tools.

Contents

This guide contains the following chapters:

- Preface, "About this Guide" introduces the organization and purpose of the Getting Started Guide, and the conventions used in this document.
- Chapter 1, "Introduction" describes the core and related information, including recommended design experience, additional resources, technical support, and submitting feedback to Xilinx.
- Chapter 2, "Licensing the Core" provides information about installing and licensing the core.
- Chapter 3, "Quick Start Example Design" provides instructions to quickly generate the core and run the example design through implementation and simulation using the default settings.
- Chapter 4, "Detailed Example Design" describes the files and directories created by the CORE Generator. It also contains detailed information about the demonstration test bench and directions for customizing it for use in a user application.
- Appendix A, "VHDL Details" provides details about the VHDL demonstration test bench and how to customize it.
- Appendix B, "Verilog Details" provides details about the Verilog demonstration test bench and how to customize it.
- Appendix C, "Data and Status Monitor Warnings" describes the common demonstration test bench warnings.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *`design_name`*
Italic font	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** `[`*`option_name`*`]` *`design_name`*
Braces { }	A list of items from which you must choose one or more	**`lowpwr`** `={`**`on`**`|`**`off`**`}`
Vertical bar |	Separates items in a list of choices	**`lowpwr`** `={`**`on`**`|`**`off`**`}`

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		.
		.
		.

Horizontal ellipsis . . .	Repetitive material that has been omitted	**`allow block`** *`block_name loc1 loc2 ... locn;`*

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Blue, underlined text	Hyperlink to a website (URL)	Go to www.xilinx.com for the latest speed files.

Chapter 1

Introduction

The LogiCORE IP SPI-4.2 (PL4) core is a fully verified design solution that supports Verilog and VHDL. The example design in this guide is provided in both Verilog and VHDL.

This chapter introduces the SPI-4.2 core and provides related information, including recommended design experience, additional resources, technical support, and how to submit feedback to Xilinx.

System Requirements

Windows

- Windows XP® Professional 32-bit/64-bit
- Windows Vista® Business 32-bit/64-bit

Linux

- Red Hat® Enterprise Linux WS v4.0 32-bit/64-bit
- Red Hat® Enterprise Desktop v5.0 32-bit/64-bit (with Workstation Option)
- SUSE Linux Enterprise (SLE) v10.1 32-bit/64-bit

Software

- ISE™ 10.1 with applicable service pack

Check the release notes for the required service pack; ISE Service Packs can be downloaded from www.xilinx.com/xlnx/xil_sw_updates_home.jsp?update=sp.

About the Core

The SPI-4.2 core is a Xilinx CORE Generator™ IP core, included in the latest IP update on the Xilinx IP Center. For detailed information about the core, see the SPI-4.2 product page. For information about system requirements, installation, and licensing options, see Chapter 2, “Licensing the Core.”

Recommended Design Experience

Although the SPI-4.2 core is a fully verified solution, the challenge associated with implementing a complete design varies, depending on desired configuration and functionality. For best results, previous experience building high-performance, pipelined FPGA designs using Xilinx implementation software and user constraints files (UCF) is recommended.

Contact your local Xilinx representative for a closer review and estimate of the effort required to meet your specific design requirements.

Additional Core Resources

For detailed information and updates about the SPI-4.2 core, see the following additional documents located on the SPI-4.2 product page.

- *LogiCORE SPI-4.2 Data Sheet*
- *LogiCORE SPI-4.2 Release Notes*
- *LogiCORE SPI-4.2 User Guide*

For updates to this document, see the *LogiCORE SPI-4.2 Getting Started Guide*, also located on the Xilinx SPI-4.2 product page.

Technical Support

To obtain technical support specific to the SPI-4.2 core, visit http://support.xilinx.com/. Questions are routed to a team of engineers with expertise using the SPI-4.2 core.

Xilinx will provide technical support for use of this product as described in the *SPI-4.2 User Guide* and the *SPI-4.2 Getting Started Guide*. Xilinx cannot guarantee timing, functionality, or support of this product for designs outside the guidelines presented in this document.

Feedback

Xilinx welcomes comments and suggestions about the SPI-4.2 core and the documentation provided with the core.

Core

For comments or suggestions about the SPI-4.2 core, please submit a WebCase from http://support.xilinx.com/. Be sure to include the following information:

- Product name
- Core version number
- Explanation of your comments

Document

For comments or suggestions about this document, please submit a WebCase from http://support.xilinx.com/. Be sure to include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

Licensing the Core

This chapter provides instructions for obtaining a license for the core so that you can use the core in a design. The SPI-4.2 core is provided under the terms of the Xilinx LogiCORE Site License Agreement. This license agreement conforms to the terms of the SignOnce IP License standard defined by the Common License Consortium. Purchase of the core entitles you to technical support and access to updates for a period of one year.

Before you Begin

This chapter assumes that you have installed the core using either the CORE Generator™ IP Update installer or by performing a manual installation after downloading the core from the web. For information about installing the core, see the SPI-4.2 product page.

Before installing the core, you must have a Xilinx.com account and the ISE 10.1 software installed on your system.

To set up an account and install the ISE software:

1. Click Sign in to Access Account at the top of the Xilinx home page; then follow the instructions to create a support account.
2. Install the ISE 10.1 software with the applicable service pack.

License Options

The SPI-4.2 core provides three licensing options, described below.

Simulation-Only Evaluation

The Simulation-Only Evaluation license is provided with the Xilinx CORE Generator system. This license lets you evaluate core functionality using a provided example design. You can also use your own design and simulate the various interfaces on the core. Functional simulation is supported by a dynamically generated gate-level netlist.

Full System Hardware Evaluation

The Full System Hardware Evaluation license is available at no cost and lets you fully integrate the core into an FPGA design, place and route the design, evaluate timing, and perform back-annotated gate-level simulation using the demonstration test bench provided.

In addition, the license lets you generate a bitstream from the placed and routed design, which can then be downloaded to a supported device and tested in hardware. The core can be tested in the target device for a limited time before *timing out*. The core can be reactivated by reconfiguring the device after a time out.

You can obtain the Full System Evaluation License in one of the following ways, depending on the core:

- By registering on the Xilinx IP Evaluation page and filling out a form to request an automatically-generated evaluation license
- By contacting your local Xilinx FAE to request a Full System Hardware Evaluation license key

Click Evaluate on the SPI-4.2 core product page for information about obtaining a Full System Hardware Evaluation License.

Full

The Full license is provided when you purchase the core. This option provides full access to all core functionality both in simulation and in hardware, including:

- Gate-level functional simulation support
- Back annotated gate-level simulation support
- Full implementation support including place and route and bitstream generation
- Full functionality in the programmed device with no time-outs

Obtaining Your License

Obtaining a Simulation-Only or Full System Hardware Evaluation License

To obtain a Simulation-Only or Full System Hardware Evaluation license, do the following:

- Navigate to the SPI-4.2 product page.
- Click Evaluate.
- Select one of the following:
  - Simulation-Only Evaluation
  - Full System Hardware Evaluation

For both types of licenses, follow the onscreen instructions to both download the CORE Generator files (delivered as an IP update) and satisfy any additional requirements associated with the license type.

Obtaining a Full License

To obtain a Full license, you must purchase the core. After purchase, you will receive a letter containing a serial number. This serial number is used to register for access to the *lounge*, a secured area of the SPI-4.2 product page.

- From the product page, click Register to request access to the lounge.
- Xilinx will review your access request. Requests for access are typically granted within 48 hours. Contact Xilinx Customer Service if you need faster turnaround.
- After you receive confirmation of lounge access, click Access Lounge on the SPI-4.2 product page and log in.

Follow the instructions in the lounge to fill out the license request form; then click Submit to automatically generate the license. An email containing the license and installation instructions will be sent to you immediately.

Installing Your License File

After selecting a license option, an email is sent to your login account that includes instructions for installing your license file. In addition, information about advanced licensing options and technical support is provided.

Chapter 3

Quick Start Example Design

The quick start steps provide information to quickly generate a SPI-4.2 core, run the design through implementation with the Xilinx tools, and simulate the example design using the provided demonstration test bench. For more detailed information about this example design, see Chapter 4, "Detailed Example Design."

Overview

The SPI-4.2 example design consists of the following:

- SPI-4.2 Sink and Source core netlists
- SPI-4.2 Sink and Source core simulation models
- Example HDL wrapper (which instantiates the cores and example design)
- Customizable demonstration test bench to simulate the example design

Generating the Core

To generate a SPI-4.2 core with default values using the Xilinx CORE Generator system, do the following:

1. Start the CORE Generator system.

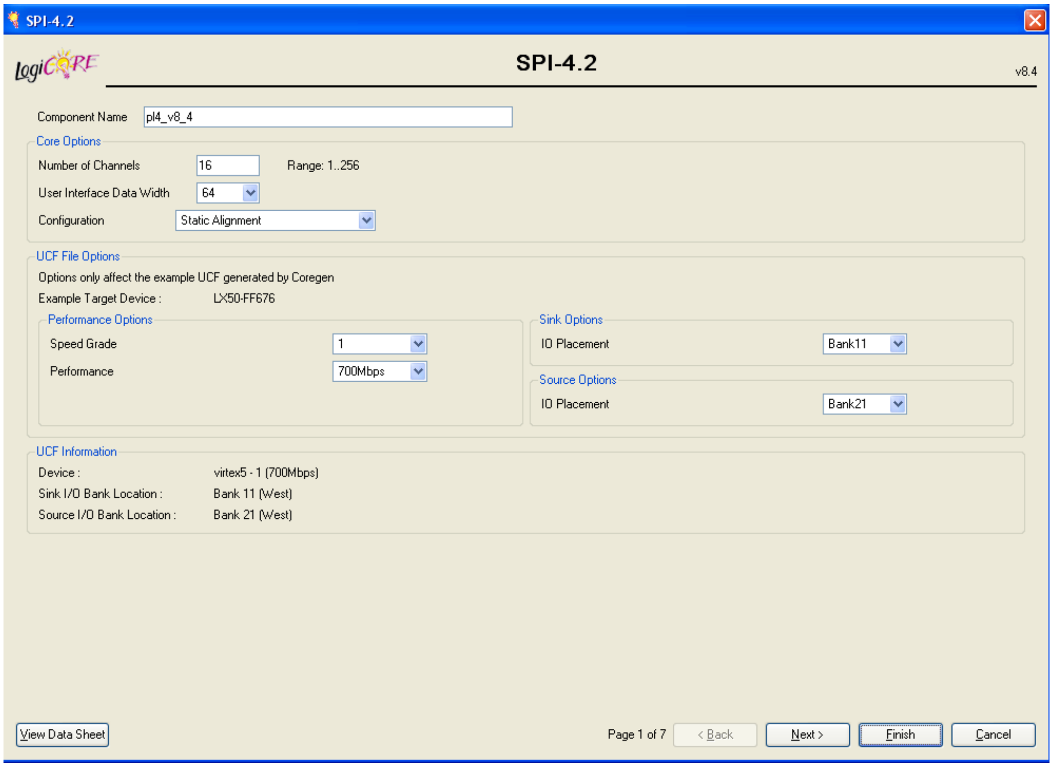
   For help starting and using the CORE Generator system, see the *Xilinx CORE Generator Guide*, available from the ISE documentation.

2. Choose File > New Project.
3. Type a directory name. For this example design, use the directory name *design*.
4. Set the following project options:
   - Part Options
     - From Target Architecture, select either Virtex™-4 or Virtex-5.

     Note: If an unsupported silicon family is selected, the SPI-4.2 core will not appear in the taxonomy tree.

     Note: The Device, Package and Speed Grade selected in the Part Options tab have no effect on the generated core. The core is delivered with an example UCF targeting either Virtex-4 4vlx25ff668 or Virtex-5 5v1x50-ff676.
   - Generation Options
     - For Design Entry, select either VHDL or Verilog.
     - For Vendor, select Synplicity or Other (for XST).

5. After creating the project, locate the directory containing the SPI-4.2 core in the taxonomy tree; it appears under Communications & Networking > Telecommunications > SPI-4.2.
6. Double-click the core to bring up the customization GUI.
7. In the Component Name field, enter a name for the core instance. (In this example, the name *quickstart* is used.)
8. After selecting the desired features and parameters from the GUI screens, click Generate.



Component Name: pl4_v8_4

Core Options
Number of Channels: 16 Range: 1..256
User Interface Data Width: 64
Configuration: Static Alignment

UCF File Options
Options only affect the example UCF generated by Coregen
Example Target Device : LX50-FF676

Performance Options
Speed Grade: 1
Performance: 700Mbps

Sink Options
IO Placement: Bank11

Source Options
IO Placement: Bank21

UCF Information
Device : virtex5 - 1 (700Mbps)
Sink I/O Bank Location : Bank 11 (West)
Source I/O Bank Location : Bank 21 (West)

Figure 3-1: **Core Customization GUI Main Window**

The cores and supporting files, including the example design, are generated in the project directory. For detailed information and an illustration of the example design files and directories produced, see "Directory and File Contents" in Chapter 4.

Implementing the Example Design

After generating a core with a Full System Hardware Evaluation or Full license, the netlists and the example design can be processed by the Xilinx implementation tools. The generated output files include scripts to assist you in running the Xilinx tools.

To implement the SPI-4.2 example design, open a command prompt or terminal window and type the following commands:

For Windows

```
ms-dos> cd <proj>\<quickstart>\implement
ms-dos> implement.bat
```

For Linux

```
% cd <proj>/<quickstart>/implement
% ./implement.sh
```

These commands execute a script that synthesizes, builds, maps, and place-and-routes the example design. The script then generates a post-par simulation model for use in timing simulation. The resulting files are placed in the results directory.

Running the Simulation

Using the provided example design, you can quickly simulate and observe the behavior of the SPI-4.2 core. There are two different simulation types, functional and timing. The simulation models provided are either in VHDL or Verilog, depending on the CORE Generator Design Entry project option selected by the user.

Setting up for Simulation

The Xilinx UniSim and SimPrim libraries must be mapped into the simulator. If the UniSim or SimPrim libraries are not set for the test environment, go to www.xilinx.com/support, where the following solution records are located:

- Compiling Xilinx Simulation Libraries (MTI) - Answer Record 2561
- Compiling Xilinx Simulation Libraries (NC-SIM) - Answer Record 2554

Functional Simulation

Instructions for running a functional simulation of the SPI-4.2 core using either VHDL or Verilog are given below. Functional simulation models are provided when the core is generated. Note that implementing the core before simulating the functional models is not required. If a configuration file (referenced in the CORE Generator GUI as the COE file) was used to program the calendar, special steps are required to include the calendar sequence in the simulation. See the *SPI-4.2 Core User Guide* for details on including the calendar initialization values in simulation.

To run a VHDL or Verilog functional simulation of the example design using MTI:

1. Set the current directory to:
   ```
   <quickstart>/simulation/functional/
   ```
2. Launch the ModelSim® simulator.
3. Launch the simulation script:
   ```
   modelsim> do simulate_mti.do
   ```

To run a VHDL or Verilog functional simulation of the example design using NCSIM:

1. Set the current directory to:

   ```
   <quickstart>/simulation/functional/
   ```

2. Execute the simulation script:

   ```
   % simulate_ncsim.sh
   ms-dos> simulate_ncsim.bat
   ```

To run a Verilog functional simulation of the example design using VCS:

1. Set the current directory to:

   ```
   <quickstart>/simulation/functional/
   ```

2. Execute the simulation script:

   ```
   % simulate vcs.sh
   ```

The simulation script compiles the functional simulation models, the loopback and the demonstration test bench, adds relevant signals to the wave window, and runs the simulation. To observe the operation of the core, inspect the simulation transcript and the waveform.

Timing Simulation

Timing simulation is available only with purchase of the core (Full license) or with access to the Full System Hardware Evaluation license. With a Simulation Only Evaluation license the core cannot be run through the implementation tools, which is required for timing based simulation.

Instructions for running a timing simulation of the SPI-4.2 core using either VHDL or Verilog are given below. A timing simulation model is generated when the core is run through the Xilinx tools using the implement script. Calendar information specified in a COE file is included in the timing simulation netlist.

To run a VHDL or Verilog l simulation of the example design using MTI:

1. Set the current directory to:

   ```
   <quickstart>/simulation/timing/
   ```

2. Launch the ModelSim simulator.
3. Launch the simulation script:

   ```
   modelsim> do simulate_mti.do
   ```

To run a VHDL or Verilog simulation of the example design using NCSIM:

1. Set the current directory to:

   ```
   <quickstart>/simulation/timing/
   ```

2. Execute the simulation script:

   ```
   ms-dos> simulate_ncsim.bat
   % simulate_ncsim.sh
   ```

To run a Verilog simulation of the example design using VCS:

1. Set the current directory to:

   ```
   <quickstart>/simulation/timing/
   ```

2. Execute the simulation script:

   ```
   % simulate_vcs.sh
   ```

The simulation script compiles the timing simulation model and the demonstration test bench, adds relevant signals to the wave window, and runs the simulation. To observe the operation of the core, inspect the simulation transcript and the waveform.

Chapter 4

Detailed Example Design

This chapter provides detailed information about the example design, including a description of files and the directory structure generated by the Xilinx CORE Generator, the purpose and contents of the provided scripts, the contents of the example HDL wrappers, and the operation of the demonstration test bench.

- **<project directory>**
  Top-level project directory; name is user-defined
  - <project directory>/<component name>
    Core release notes file
    - <component name>/doc
      Product documentation
    - <component name>/example design
      Verilog and VHDL design files
    - <component name>/implement
      Implementation script files
      - implement/results
        Results directory created after implementation scripts are run; contains implement script results.
    - <component name>/simulation
      Simulation scripts
      - simulation/functional
        Functional simulation files
      - simulation/timing
        Timing simulation files

Directory and File Contents

The SPI-4.2 core directories and their associated files are defined in the following sections.

<project directory>

The project directory contains all the CORE Generator project files. See the *SPI-4.2 User Guide* for detailed information about each file.

Table 4-1: **Project Directory**

Name	Description
`<project_dir>`	
`<component_name>_pl4_snk_top.ngc` `<component_name>_pl4_src_top.ngc`	Top-level netlists.
`<component_name>_pl4_snk_top.v[hd]` `<component_name>_pl4_src_top.v[hd]`	Verilog and VHDL simulation models.
`<component_name>.xco`	CORE Generator project-specific option file; can be used as an input to the CORE Generator.
`<component_name>_flist.txt`	List of files delivered with the core.
`<component_name>_pl4_snk_top.{vho|veo}` `<component_name>_pl4_src_top.{vho|veo}`	VHDL and Verilog instantiation templates.

Back to Top

<project directory>/<component name>

The <component name> directory contains the release notes file provided with the core, which may include last-minute changes and updates.

Table 4-2: **Component Name Directory**

Name	Description
`<project_dir>/<component_name>`	
`spi4_2_readme.txt`	Core release notes text file.

Back to Top

<component name>/doc

The doc directory contains the PDF documentation provided with the core.

Table 4-3: **Doc Directory**

Name	Description
`<project_dir>/<component_name>/doc`	
`spi4_2_ds209.pdf`	*SPI-4.2 Data Sheet*
`spi4_2_gsg154.pdf`	*SPI-4.2 Getting Started Guide*

Table 4-3: **Doc Directory** ***(Continued)***

Name	Description
`spi4_2_ug153.pdf`	*SPI-4.2 User Guide*

Back to Top

<component name>/example design

The example design directory contains the example design files provided with the core.

Table 4-4: **Example Design Directory**

Name	Description
`<project_dir>/<component_name>/example_design`	
`<component_name>_top.ucf`	User constraints file (UCF) provides example constraints necessary for processing the core using Xilinx implementation tools. This file can be modified to meet individual system requirements. The example UCF contains timing and placement constraints for both Sink and Source cores.
`<component_name>_top.v[hd]`	VHDL or Verilog wrapper file for the example design; it instantiates the Sink and Source cores and the loopback module. This is the top-level synthesis file for the example design.
`pl4_fifo_loopback.v[hd]`	Top-level loopback file used in the example design; it instantiates the loopback read and write modules.
`pl4_fifo_loopback_read.v[hd]`	Loopback read module used in the example design; it interfaces to the SPI-4.2 Sink core.
`pl4_fifo_loopback_write.v[hd]`	Loopback write module used in the example design; it interfaces to the SPI-4.2 Source core.
`pl4_src_clk.v[hd]`	Example clocking module used in the example design when the Source core is configured for slave clocking.
`virtex4.v`	Module instantiation for Virtex-4 primitives
`virtex5.v`	Module instantiation for Virtex-5 primitives

Back to Top

<component name>/implement

The implement directory contains the core implementation script files.

Table 4-5: **Implement Directory**

Name	Description
`<project_dir>/<component_name>/implement`	
`implement.{sh|bat}`	Windows (.bat) or Linux (.sh) script that processes the example design through the Xilinx tool flow.
`xst.prj`	XST project file for the example design; it lists all of the source files to be synthesized. It is only available when the CORE Generator vendor project option is set to "Other."
`xst.scr`	XST script file for the example design that is used to synthesize the core, and it is called from implement.{sh|bat}. It is only available when the CORE Generator vendor project option is set to "Other."
`synplify.prj`	Synplicity project file for the example design; it lists all of the source files to be synthesized. It is only available when the CORE Generator vendor project option is set to "Synplicity."

Back to Top

implement/results

The results directory is created by the implement script, after which the implement script results are placed in the results directory.

Table 4-6: **Results Directory**

Name	Description
`<project_dir>/<component_name>/implement/results`	
Implement script result files.	

Back to Top

<component name>/simulation

The simulation directory contains the necessary files to test a VHDL or Verilog example design with the demonstration test bench.

Table 4-7: **Simulation Directory**

Name	Description
`<project_dir>/<component_name>/simulation`	
`data_file.dat`	Data file containing the data to be sent across the SPI-4.2 Interface
`pl4_clk_gen.v[hd]`	Demo Test bench Clock Generator
`pl4_data_monitor.v[hd]`	Demo Test bench Data Monitor
`pl4_demo_testbench.v[hd]`	Demo Test bench Top Level Module
`pl4_procedures.v[hd]`	Demo Test bench Procedures Module
`pl4_src_clk.v[hd]`	HDL file which is utilized if the Slave core is configured with slave clocking
`pl4_startup.v[hd]`	Demo Test bench DCM Startup and Calendar Loader Module
`pl4_status_monitor.v[hd]`	Demo Test bench Status Monitor
`pl4_stimulus.v[hd]`	Demo Test bench Data and Status Stimulus Module
`pl4_testcase.v[hd]` `pl4_testcase_pkg.v[hd]`	Controls the operation of the demonstration test bench and can be user-modified.
`snk_calendar.dat`	Data file containing the calendar information for the Sink interface
`src_calendar.dat`	Data file containing the calendar information for the Source interface
`[glbl.v]`	Asserts initial global reset pulse (Verilog only)

Back to Top

simulation/functional

The functional directory contains functional simulation scripts provided with the core.

Table 4-8: **Functional Directory**

Name	Description
`<project_dir>/<component_name>/simulation/functional`	
`simulate_mti.do`	ModelSim macro file that compiles the functional netlist, loopback HDL, and demo HDL source. The script also loads and runs the simulation for 8 µs.
`wave_mti.do`	ModelSim macro file that opens a wave window and adds key signals to the wave viewer. The wave_mti.do file is called by the simulate_mti.do macro file.
`simulate_ncsim.sh` `simulate_ncsim.bat`	Shell scripts that compile the functional netlist and loopback HDL source. The script also launches NCSIM and runs the simulation for 8 µs.
`wave_ncsim.sv`	NCSIM macro file that opens a wave window and adds key signals to the wave viewer. The wave_ncsim.sv file is called by the simulate_ncsim.sh or simulate_ncsim.bat file.
`simulate_vcs.sh (verilog only)`	Shell script that compiles the functional netlist and example design. The script also runs the functional simulation using VCS.
`vcs_session.tcl (verilog only)`	VCS tcl script that opens a wave window. This macro is called by the simulate_vcs.sh script.
`vcs_commands.key (verilog only)`	VCS command file. This file is called by the simulate_vcs.sh script.

Back to Top

simulation/timing

The timing directory contains timing simulation scripts provided with the core.

Table 4-9: **Timing Directory**

Name	Description
`<project_dir>/<component_name>/simulation/timing`	
`simulate_mti.do`	ModelSim macro file that compiles the post-par timing netlist and demo HDL source. The script also loads and runs the simulation for 8 µs. The implement script must first be run to generate the post-par timing simulation model. Simulation can only be run after the timing simulation model is generated.
`wave_mti.do`	ModelSim macro file that opens a wave window and adds key signals to the wave viewer. The wave_mti.do file is called by the simulate_mti.do macro file.
`simulate_ncsim.sh` `simulate_ncsim.bat`	Shell scripts that compile the functional netlist and loopback HDL source. The script also launches NCSIM and runs the simulation for 8 µs.
`wave_ncsim.sv`	A NCSIM macro file that opens a wave window and adds key signals to the wave viewer. The wave_ncsim.sv file is called by the simulate_ncsim.sh or simulate_ncsim.bat file.
`simulate_vcs.sh (verilog only)`	Shell script that compiles the structural netlist and example design. The script also runs the functional simulation using VCS.
`vcs_session.tcl (verilog only)`	VCS tcl script that opens a wave window. This macro is called by the simulate_vcs.sh script.
`vcs_commands.key (verilog only)`	VCS command file. This file is called by the simulate_vcs.sh script.

Back to Top

Implementation and Simulation Scripts

The implementation script is either a shell script or a batch file that runs the example design through the Xilinx tool flow. The scripts are located in the following directory:

`<proj_dir>/<component_name>/implement/`

The implementation scripts are parameterized based on the Design Entry Tool and Design Entry Language CORE Generator project options. If either of these project options are changed, the core must be regenerated to create the appropriate implementation scripts.

If the core was generated with the Full System Hardware Evaluation or the Full license, the implementation script is present and performs the following steps:

1. Synthesizes the example design using the selected synthesis tool (XST or Synplify).
2. Runs `ngdbuild` to consolidate the core netlists, wrapper netlist, and constraints file into the common database.
3. Runs `map` to perform technology specific mapping of the design.
4. Runs `par` to perform place and route of the design.
5. Runs `trce` to perform static timing analysis of the routed design.
6. Runs `bitgen` to generate a bitstream for download to the target FPGA.
7. Runs `netgen` to generate a post-par simulation model for use in timing simulation.

Simulation Script Details

The simulation scripts for ModelSim and NCSIM that simulate the demonstration test bench are located in one of the following directories:

`<proj_dir>/<component_name>/simulation/{functional | timing }/`

For functional simulation, the simulation script performs the following tasks:

1. Compiles the simulation models provided with the core.
2. Compiles the loopback example design.
3. Compiles the wrapper file, which instantiates the cores and the loopback.
4. Compiles the demonstration test bench.
5. Starts a simulation of the demonstration test bench.
6. Opens the waveform viewer and adds key signals (`wave_mti.do|wave_ncsim.sv`).
7. Runs the simulation.

For timing simulation, the simulation script performs the following tasks:

1. Compiles the post-par design example, which includes the cores and the loopback.
2. Compiles the demonstration test bench.
3. Starts a simulation of the demonstration test bench.
4. Opens the waveform viewer and adds key signals (`wave_mti.do|wave_ncsim.sv`).
5. Runs the simulation.

Example Design Configuration

In the example design, a Loopback Module is connected to the user interface of the SPI-4.2 core. Typically, the user interface would be connected directly to the design. The SPI-4.2 Interface, which is the interface defined by the *OIF-SPI4-02.1* specification, typically connects to a SPI-4.2 PHY layer device or network processor. Figure 4-1 shows the example design modules architecture and interfaces to the SPI-4.2 core.

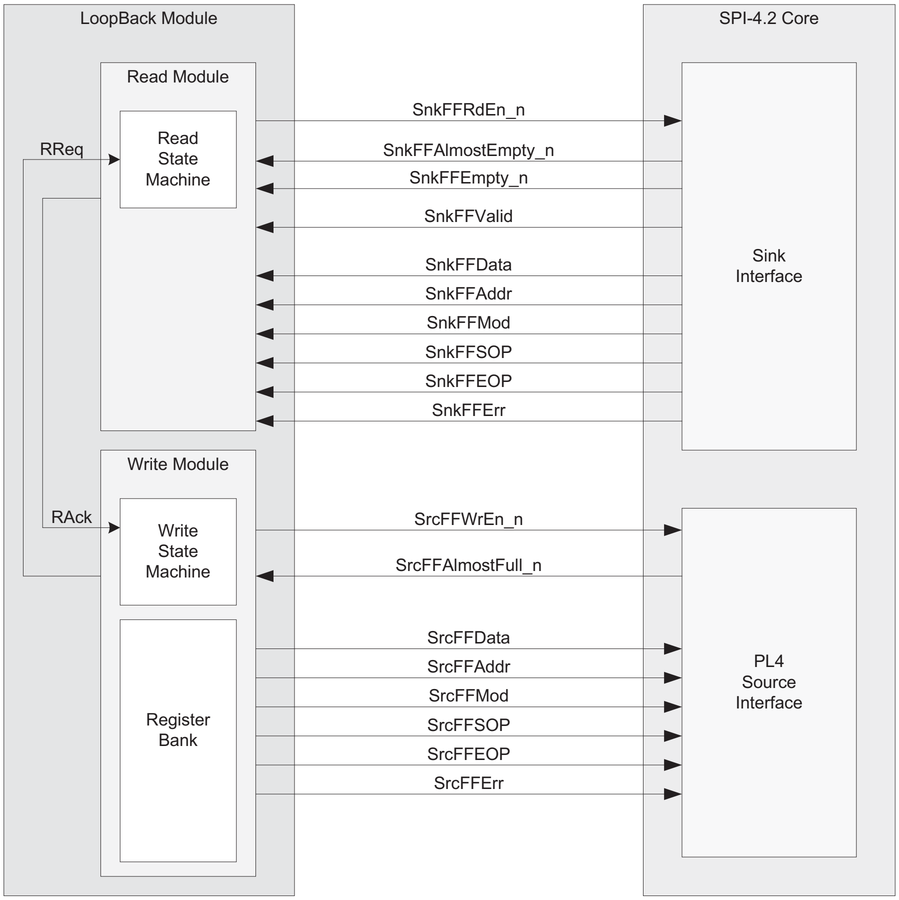

LoopBack Module
Read Module
Read State Machine
RReq
RAck
Write Module
Write State Machine
Register Bank
SPI-4.2 Core
Sink Interface
PL4 Source Interface
SnkFFRdEn_n
SnkFFAlmostEmpty_n
SnkFFEmpty_n
SnkFFValid
SnkFFData
SnkFFAddr
SnkFFMod
SnkFFSOP
SnkFFEOP
SnkFFErr
SrcFFWrEn_n
SrcFFAlmostFull_n
SrcFFData
SrcFFAddr
SrcFFMod
SrcFFSOP
SrcFFEOP
SrcFFErr

Figure 4-1: **Example Design Configuration**

Loopback Module

The Loopback Module connects to the user interface of the SPI-4.2 Sink and Source cores. There is a Read Module that accesses packet data from the Sink FIFO and a Write Module that transfers data into the Source FIFO. The Read Module polls the status signals `SnkFFEmpty_n` and `SnkFFAlmostEmpty_n` to determine whether it can perform a read from the Sink FIFO. The Write Module polls `SrcFFAlmostFull_n` to determine whether it can transfer data into the Source FIFO.

Basic Loopback Operation

When the Almost Full flag (`SrcFFAlmostFull_n`) is deasserted, the Write Module asserts a read request (`RReq`) that is sent to the Read Module. When a read request is received, the Read Module verifies that the FIFO is not empty and initiates a read from the Sink FIFO. On the next cycle, the data appears on `SnkFFData`, and `SnkFFValid` is asserted. `SnkFFValid` drives the `SrcFFWrEn_n` signal directly, which enables the writing of data into the Source FIFO. The transfer of data continues until the Source FIFO becomes almost full or the Sink FIFO becomes empty. If the Source FIFO becomes almost full, all outstanding data is written into the Source FIFO and the transfer of data between the FIFOs is halted.

Demonstration Test Bench

The demonstration test bench emulates a PHY device by generating and receiving packet data across the SPI-4.2 interface. The interface between the demonstration test bench and the SPI-4.2 core is illustrated in Figure 4-2.

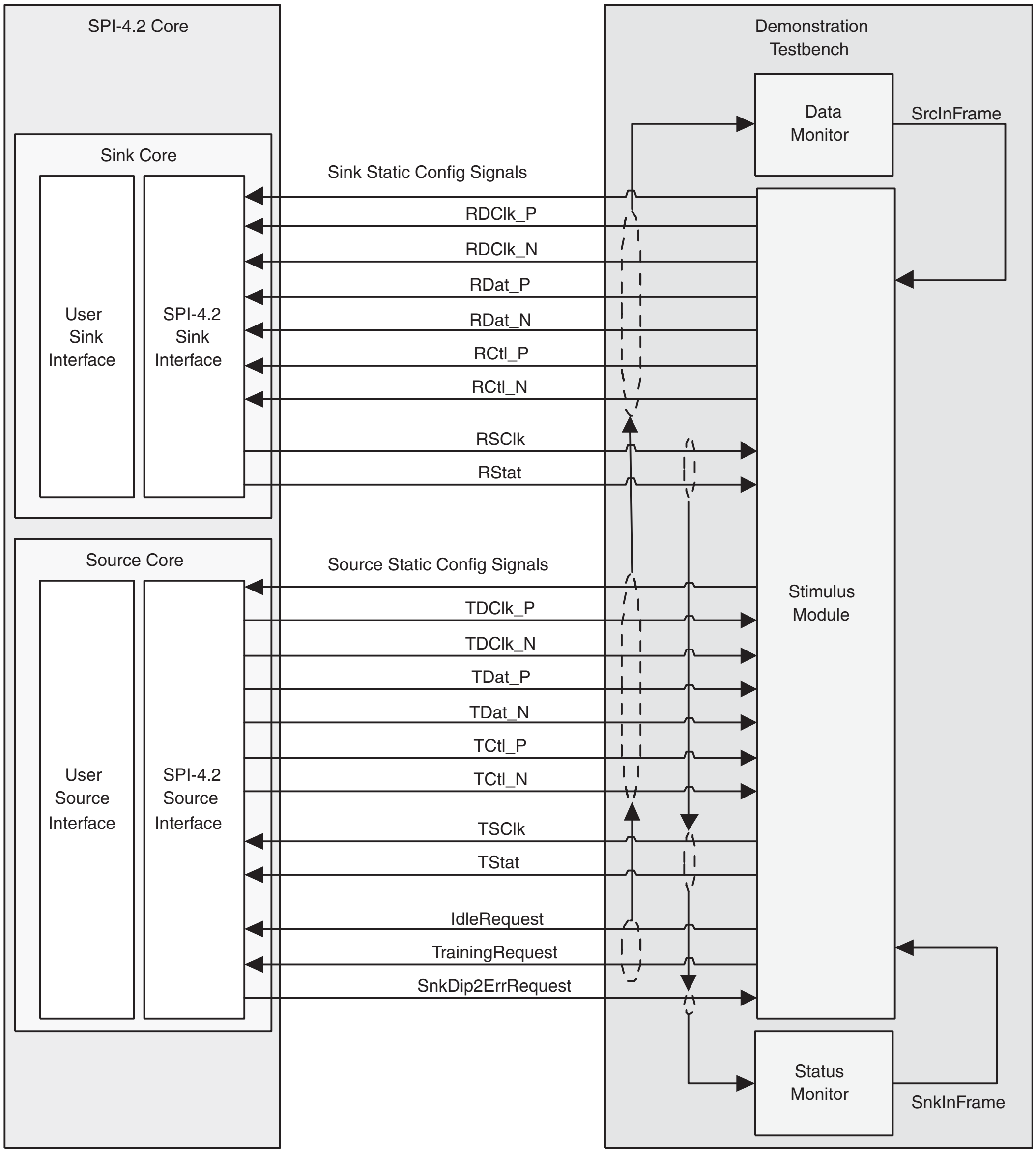

SPI-4.2 Core
Sink Core
User Sink Interface
SPI-4.2 Sink Interface
Source Core
User Source Interface
SPI-4.2 Source Interface
Demonstration Testbench
Data Monitor
SrcInFrame
Stimulus Module
Status Monitor
SnkInFrame
Sink Static Config Signals
RDClk_P
RDClk_N
RDat_P
RDat_N
RCtl_P
RCtl_N
RSClk
RStat
Source Static Config Signals
TDClk_P
TDClk_N
TDat_P
TDat_N
TCtl_P
TCtl_N
TSClk
TStat
IdleRequest
TrainingRequest
SnkDip2ErrRequest

Figure 4-2: **Demonstration Test Bench Connections**

The modules for sending data and status are described in "Customizing the Demonstration Test Bench," later in this section. As described below and shown in Figure 4-3, the demonstration test bench consists of the following modules:

- Clock Generator
- Startup
- Stimulus
- Data Monitor

- Status Monitor
- Testcase

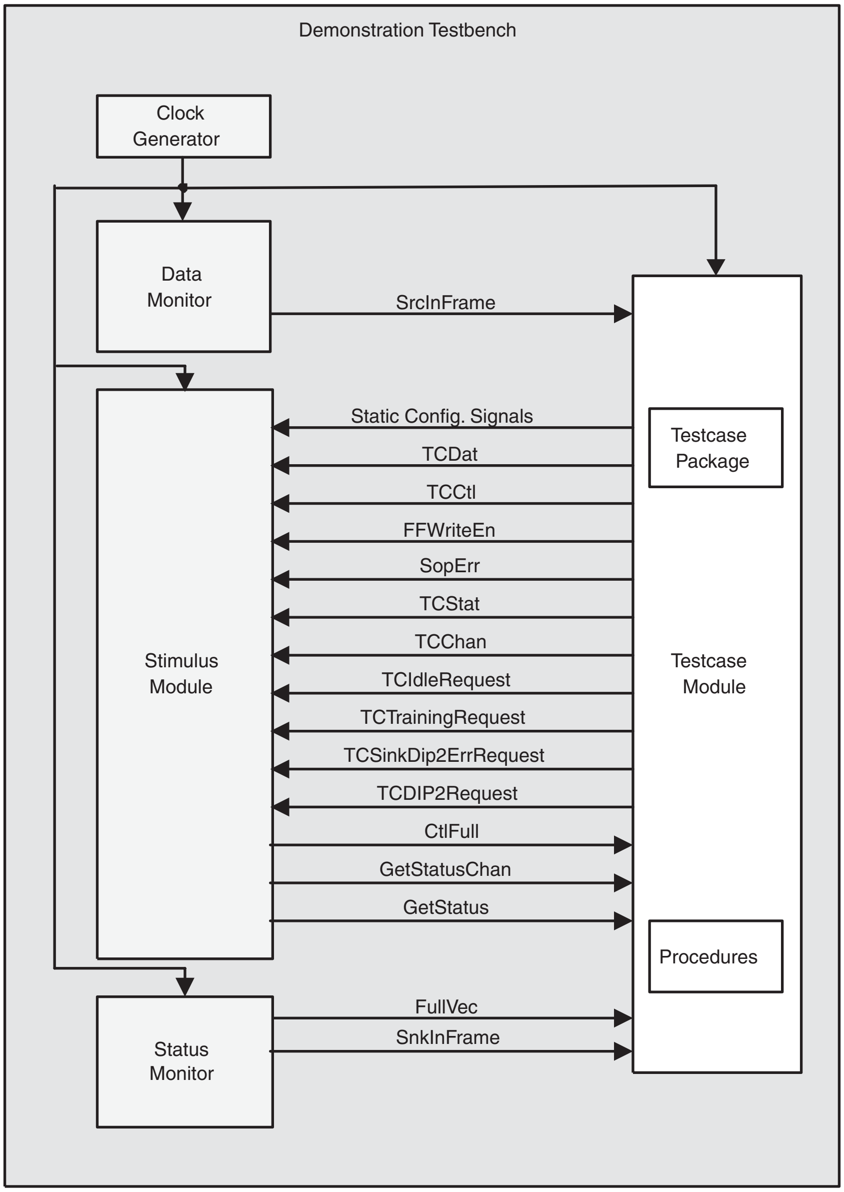

Demonstration Testbench
Clock Generator
Data Monitor
SrcInFrame
Stimulus Module
Static Config. Signals
TCDat
TCCtl
FFWriteEn
SopErr
TCStat
TCChan
TCIdleRequest
TCTrainingRequest
TCSinkDip2ErrRequest
TCDIP2Request
CtlFull
GetStatusChan
GetStatus
Testcase Package
Testcase Module
Procedures
Status Monitor
FullVec
SnkInFrame

Figure 4-3: **Test Bench Modules**

Clock Generator

The Clock Generator creates all of the clocks that are used in the Design Example, including `SysClk`, `RDClk2x`, `UserClk`, `TSClk`, and `SnkIdelayRefClk`. These clocks are described in more detail in Table 4-10.

Startup Module

The Startup Module contains three functions: DCM setup, calendar loading, and Dynamic Phase Alignment (DPA) Initialization. These functions are described in detail in the following sections.

DCM Startup

The DCM Startup is a state machine that ensures that the DCMs are reset in the appropriate order. If they are not reset appropriately, the DCMs will not lock. The Startup Module first asserts `DCMReset_TDClk`. Once `Locked_TDClk` is asserted, it resets `DCMReset_RDClk`. Then it waits for `Locked_RDClk` before asserting `DCMReset_TSClk`. After `Locked_TSClk` is asserted, the state machine waits until the `SnkClksRdy` and `SrcClksRdy` signals are asserted. The `Reset_n` signal is deasserted only after this occurs. All operations are performed in the `SysClk` domain.

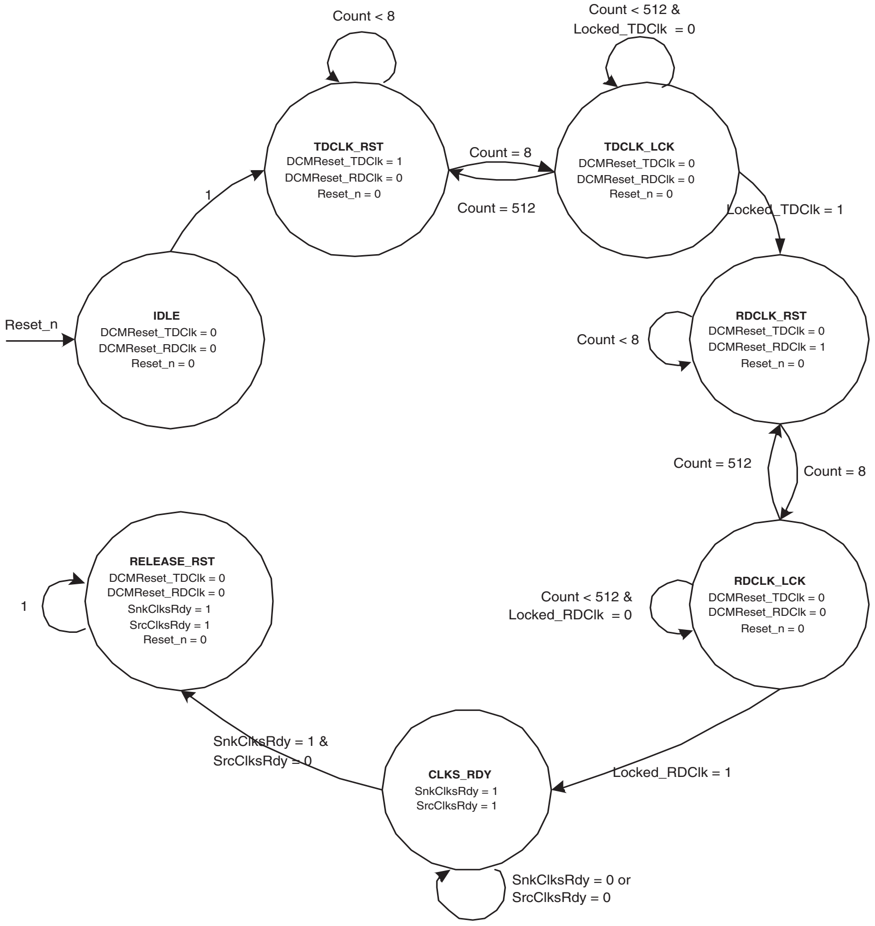

Count < 8
Count < 512 &
Locked_TDClk = 0
TDCLK_RST
DCMReset_TDClk = 1
DCMReset_RDClk = 0
Reset_n = 0
Count = 8
Count = 512
TDCLK_LCK
DCMReset_TDClk = 0
DCMReset_RDClk = 0
Reset_n = 0
Locked_TDClk = 1
1
Reset_n
IDLE
DCMReset_TDClk = 0
DCMReset_RDClk = 0
Reset_n = 0
Count < 8
RDCLK_RST
DCMReset_TDClk = 0
DCMReset_RDClk = 1
Reset_n = 0
Count = 512
Count = 8
RELEASE_RST
DCMReset_TDClk = 0
DCMReset_RDClk = 0
SnkClksRdy = 1
SrcClksRdy = 1
Reset_n = 0
1
Count < 512 &
Locked_RDClk = 0
RDCLK_LCK
DCMReset_TDClk = 0
DCMReset_RDClk = 0
Reset_n = 0
SnkClksRdy = 1 &
SrcClksRdy = 0
CLKS_RDY
SnkClksRdy = 1
SrcClksRdy = 1
Locked_RDClk = 1
SnkClksRdy = 0 or
SrcClksRdy = 0

Figure 4-4: **Startup State Diagram**

Figure 4-4 illustrates the nine states for this machine.

- **IDLE** Initial state after reset; `DCMReset_TDClk` is asserted.
- **TDCLK_RST** Holds `DCMReset_TDClk` for 8 cycles then releases it.
- **TDCLK_LCK** Waits for the `Locked_TDClk` signal.

- **RDCLK_RST** Holds `DCMReset_RDClk` for 8 cycles then releases it
- **RDCLK_LCK** Waits for the `Locked_RDClk` signal.
- **TSCLK_RST** Holds `DCMReset_TSClk` for 12 cycles then releases it.
- **TSCLK_LCK** Waits for the `Locked_TSClk` signal.
- **CLKS_RDY** Waits for `SnkClksRdy` and `SrcClksRdy` signals.
- **RELEASE_RST** Releases `Reset_n`.

Calendar Loader

The second function of the Startup module is the logic to load the calendars. The demonstration test bench reads the Sink calendar sequence and the Source calendar sequence from two different files and loads this information into the calendars of the Sink and Source cores and into the Stimulus module. It also loads the calendar into the Status Monitor so that it can identify which channel is receiving status. The calendar sequences can be modified (see "Calendar Sequence Files (Sink and Source)," page 43).

DPA Initialization

The third function of the Startup module is to initialize the Dynamic Phase Alignment section of the Sink core. It is present in the module only if Dynamic Alignment is selected in the CORE Generator system. It simply asserts the PhaseAlignRequest signal to the Sink core for two cycles of `UserClk` once the core is out of reset.

Once `PhaseAlignRequest` is asserted, the dynamic alignment algorithm needs some time before completing its alignment and asserting `PhaseAlignComplete`. This value is dependent on the frequency of `RDClk` and when `PhaseAlignRequest` is asserted.

Stimulus Module

While the testcase and procedures modules are used to generate data and status, the stimulus module is used to actually send this data to the SPI-4.2 core. The stimulus module either transmits data and status generated by the testcase module, or it directly transmits training or idle data and framing status. In addition to sending status and data, the stimulus module drives the static configuration signals defined in the testcase module. The behavior of the stimulus module can be modified with the constants defined in the testcase package.

The Stimulus module also performs the following operations:

- Sends training or framing if the core is out of frame
- Inserts periodic training on `RDat`
- Ensures minimum SOP spacing is met
- Calculates DIP2 and DIP4 values
- Drives Source core request signals
- Merges SOP and EOP control words

The Stimulus module has two status inputs: `SnkInFrame` and `SrcInFrame`. If `SnkInFrame` is deasserted, the stimulus module sends training patterns over `RDat` until `SnkInFrame` is asserted. If `SrcInFrame` is deasserted, the stimulus module sends framing over `TStat` until `SrcInFrame` is asserted.

Procedures Module

The procedures module is a package of functions instantiated in the testcase module to simplify sending data and status to the stimulus module. Using these functions, you can create any desired sequence of data or status. The method by which functions are called varies among languages, and is described in the appendices.

The following functions are supported in the procedures module:

- **send_packet** Used to transmit an entire packet of data. This procedure will always send an SOP control word before the burst of data and an EOP control word following the data burst.
- **send_user_data** Used to transmit a burst of data. The presence of an SOP control word (before the burst of data) and an EOP control word (following the data burst) can be specified. The EOP can optionally specify an abort (ERR).
- **send_idles** Used to send idle cycles.
- **send_training** Used to send training patterns.
- **sop_spacing** Used to send erred data by sending two SOP words in less than eight cycles. This function limits the number of cycles between the two SOPs to less than seven. This ensures that an SOP spacing error occurs.
- **reset** Used to reset the interface to the stimulus module. Should be called at the beginning of any testcase.
- **send_status** Used to change the status (on `TStat`) for a particular channel.
- **get_status** Used to check the status of a specific channel.

Data Monitor

The data monitor is responsible for verifying that data sent from the demonstration test bench is the same as the data received from the core. This is accomplished by monitoring the `RDat` and `RCtl` signals that are input into the Sink core, and comparing them to the `TCtl` and `TDat` signals output from the Source core. This is a simple comparison as long as the data being sent does not violate the *OIF-SPI4-02.1* specification. If the specification is violated, the SPI-4.2 core modifies the data to enforce compliance, and the data monitor accounts for the modification before comparing `TDat` to `RDat`. In addition to the data, the monitor also verifies DIP4, SOP spacing, IDLE request, Training request, `DATA_MAX_T`, and `ALPHA_DATA` compliance. Changes in the testcase can create situations that cause the data monitor to output warning messages. For more information on output warning messages, see Appendix C, "Data and Status Monitor Warnings."

Status Monitor

The status monitor inspects the `RStat` bus. In addition to verifying correct values for channel status, it compiles the current status for each channel into the vector `FullVec`. `FullVec` is used by the testcase module when the `CHECK_RSTAT` constant is set to stall data on `RDat` when the targeted channel is full. See Table 4-11 for more information about the `FullVec` vector.

The status monitor also calculates the DIP2 value for `RStat` and compares it with what is actually received. If there is an error, it looks at the signal `SnkDIP2ErrRequest` to see if it was asserted and the error is expected.

Lastly, the signal `SnkInFrame` is created in the status monitor by inverting `SnkOof`. This signal is used by the stimulus module to send training. See Appendix C, "Data and Status Monitor Warnings."

Customizing the Demonstration Test Bench

The demonstration test bench can be used with default settings or customized to observe the behavior of the SPI-4.2 core for different configurations.

The demonstration test bench can be programmed to transmit a range of stimuli by modifying `TSCLK_LCK`.

- Testcase Package—contains constants used by the testcase module
- Testcase Module—generates data and status
- Sink Calendar Sequence—contains the channel order for the Sink core status
- Source Calendar Sequence—contains the channel order for the Source core status

The following sections describe each module, including customization methods and resulting behavior. The module descriptions are applicable to both VHDL and Verilog designs. Language-specific details for VHDL are provided in Appendix A, "VHDL Details." Language-specific details and source code showing how to further randomize input to the SPI-4.2 core for Verilog are provided in Appendix B, "Verilog Details."

Test Case Package

The test case package contains a list of constants that define the ways that the cores and demonstration test bench operate. Some of these are user-defined and can be modified, while others are defined when the core is generated. Table 4-10 provides test bench constants that can be modified. These constants are modified by regenerating the core in the CORE Generator system.

Table 4-10: **Testcase Package User-Defined Constants**

Name	Constant Type	Default Value (Range)	Description
SNK_CAL_DATA	String	snk_calendar.dat <filename>	Contains the name of the file with the Sink calendar sequence to be programmed.
SRC_CAL_DATA	String	src_calendar.dat <filename>	Contains the name of the file with the Source calendar sequence to be programmed.
SNK_ALPHA_DATA	Integer	3 <0 - 255>	Sets the number of repetitions of the 20-word training pattern sent to the Sink core (0 means don't send periodic training).
SNK_DATA_MAX_T	Integer	4000 <0-65535>	Sets the number of cycles between training patterns sent to the Sink core (0 means don't send periodic training).

Table 4-10: **Testcase Package User-Defined Constants *(Continued)***

Name	Constant Type	Default Value (Range)	Description
MERGE_PAYLOAD	Integer	0 <0 or 1>	Before data is sent on RDat, the demonstration test bench can either merge an EOP and SOP control word into one payload control word, or it can leave them as two separate control words. **1**: Merge EOP and SOP is enabled. **0**: Merge EOP and SOP is disabled.
CHECK_RSTAT	Integer	0 <0 or 1>	The demonstration test bench can operate in two modes with respect to the incoming status signal RStat. It either ignores the value on RStat or checks the value on RStat. **0**: Ignore the value on RStat. The test bench continues to send data on RDat regardless of the status of the current channel. **1**: Check the value on RStat. The test bench checks the status of the current channel before sending data to it. If the channel is satisfied (RStat = '10'), then the test bench does not send the packet of data and instead tries to send the next packet. The test bench sends the packet if the channel is starving or hungry (RStat = '01' or '00').
DATA_TYPE	Integer	1 <0, 1, 2>	Three types of data can be generated on RDat. The first type simply increments the data on each channel (e.g. sends 0, 1, 2 to channel 0, sends 0, 1, 2 to channel 1, then sends 3, 4, 5 to channel 0). The second sends randomized data on RDat. The last type sends data read from the file <TEST_DATA_FILE>. **0**: Send incremental data **1**: Send random data **2:** Send data read from file
TEST_DATA_FILE	String	data_file.dat <filename>	Contains the name of the file to be read if DATA_TYPE = 2
RANDOM_SEED	Integer (Verilog)	5431 <any 32-bit integer value>	Initial seed for the random number generator. To get different results between two runs of a random test bench, the seed must be changed. If the seed is not changed between runs, then every random number is the same as the previous run.
	std_logic_vector(31 downto 0) (VHDL)	x"1537" <any 32-bit vector>	

Table 4-10: **Testcase Package User-Defined Constants *(Continued)***

Name	**Constant Type**	**Default Value (Range)**	**Description**
DATA_NUM_TRAIN_SEQ	Integer	3 <0 - 255>	Sets the number of complete training patterns that the demonstration test bench has to receive on TDat (upon startup) before it stops sending framing sequences on TStat. Once this happens, the demonstration test bench begins sending valid status.
TDCLK_PERIOD	Time	2.86 ns <time>	Sets the period of the SysClk signal, which is used by the Source core to generate TDClk. Value must be greater than or equal to 2.00 ns (≤ 500 MHz).
RDCLK_PERIOD	Time	2.86 ns <time>	Sets the period of the RDClk signal and the half-period of the RDClk2x signal. Value must be greater than or equal to 2.00 ns (≤ 500 MHz).
USERCLK_PERIOD	Time	5.71 ns <time>	Sets the period of the UserClk, used for the loopback interface to the cores and programming of the calendars. Value must be greater than or equal to 4.00 ns (≤ 250 MHz).
TFF	Time	500 ps <time>	Clock-to-out time used by logic in the demonstration test bench

Testcase Module

The testcase module generates data and sends it to the stimulus module, which in turn transmits data to the Sink core and status to the Source core. The following data is created in the testcase module:

- Static configuration signals
- SPI-4.2 and demonstration test bench requests
- Source core status and Sink core data

Figure 4-2 shows the interface between the testcase and stimulus modules.

The static configuration signals are set when the SPI-4.2 core is generated; these signals can also be modified in circuit. The description of these signals can be found in the *SPI-4.2 Core User Guide*.

The status and data generation is simplified by instantiating the procedures module and calling the functions contained in the module. This allows the testcase module to be completely asynchronous, as all of the clocking is done in the procedures module.

Table 4-11 contains a list of common useful test case signals and descriptions.

Table 4-11: **Useful Testcase Signals**

Name	Description
FullVec	An array of bits indicating the last status received on RStat for each channel. For each channel, the corresponding bit is set (1) if the status received was '10' - satisfied, and cleared (0) if the status was '01' - hungry or '00' - starving.
NumLinks	The number of channels for which the core was configured.
Reset_n	Reset signal to the Sink and Source core (active low).
SnkEn	Enable signal to the Sink core.
SnkFifoReset_n	FIFO Reset signal to the Sink core (active low).
SnkInFrame	Asserted when the Sink core is in frame (as interpreted by the status monitor).
SnkOof	Out-of-Frame signal from the Sink core.
SrcEn	Enable signal to the Source core.
SrcFifoReset_n	FIFO Reset signal to the Source core (active low).
SrcInFrame	Asserted when the Source core is in frame (as interpreted by the data monitor).
SrcOof	Out-of-Frame signal from the Source core.

There are five request signals that can be asserted in the testcase module. The first four signals interface to the stimulus module (see Figure 4-2, page 34). The fifth is encapsulated with the generated data sent to the stimulus module. Table 4-12 details request signals.

Table 4-12: **Testcase Module Request Signals**

Name	Function
TCIdleRequest	Drives the IdleRequest input to the Source core, which results in idles begin transmitted on TDat.
TCTrainingRequest	Drives the TrainingRequest input to the Source core, which causes training to be sent on TDat.
TCSnkDip2ErrRequest	Drives the SnkDip2ErrRequest input to the Sink core, which results in DIP2 errors on RStat.
TCDIP2Request	When asserted (active high), causes DIP2 errors to be transmitted on TStat.
TCDIP4Request	When asserted (active high), causes DIP4 errors to be transmitted on RDat.

In addition to the request signals described above, the test case module has control over the Sink and Source cores with the `SnkEn`, `SrcEn`, `SnkFifoReset_n`, and `SrcFifoReset_n` signals. Descriptions of these signals can be found in the *SPI-4.2 Core User Guide*.

The Source core status is also generated in the test case module using functions contained in the procedures module. Using the function `send_status`, you can specify a channel and the status for that channel. This sends the status and the channel to the stimulus module for transmission to the core. The stimulus module ensures that the status is sent in the correct location of the calendar sequence.

Calendar Sequence Files (Sink and Source)

The `snk_calendar.dat` and `src_calendar.dat` files are used to define the order that status is sent on the SPI-4.2 Interface. The number of lines in a file is equal to the length of the calendar sequence (`SnkCalendar_Len + 1` and `SrcCalendar_Len +1`). Each line of the file represents an 8-bit calendar entry in hexadecimal format. For example, a calendar with a length of five and a sequence of <channel 0, channel 1, channel 0, channel 2, channel 3> can be generated by the following format:

```
00
01
00
02
03
```

File names are defined in the test case package, and can be changed if desired.

Appendix A

VHDL Details

Procedures Module

The procedures module is a package of functions instantiated in the testcase module to simplify the sending of data and status to the Stimulus module. By using these functions, the user can create any desired sequence of data or status. All functions are called from the Testcase module using the following format:

```
Format: <function name>(<IOBus>, <inputs>)
Example: send_packet(ProBusR, 0, 40): A 40-byte long packet is sent on
channel 0.
```

The procedures module handles all clocking for the Testcase module. For an example of how these procedures are used, see the default file (`pl4_testcase.vhd`) provided with the core.

All functions in the VHDL procedures module use a passed-in record to inspect and modify the state of the interface with the Stimulus module. There are two such record types defined in the procedures module: `ProceduresRDClkBusType` (PBr) and `ProceduresTSClkBusType` (PBt). For a usage example, see the provided testcase file (`pl4_testcase.vhd`).

The tables in this section describe supported functions included in the procedures module.

reset (`PBr`) and **reset** (`PBr`) procedures are used to initialize the PBr and PBt records. They must be called at the beginning of every testcase.

The **send_packet** procedure is used to transmit an entire packet of data. This procedure always sends a SOP control word before the burst of data and an EOP control word following the data burst. The EOPs (bits 14:13 of the control word following the burst) are automatically calculated from the number of bytes sent.

Table A-1: **send_packet (PBr, addr, bytes) Inputs**

Name	Range	Description
ADDR	0 to 255	Channel on which the packet should be sent.
BYTES	1 to 255	Number of bytes to send on the selected channel.

The **send_user_data** procedure is used to transmit a burst of data. The presence of a SOP control word (before the burst of data) and an EOP control word (following the data burst), can be specified. The EOPs (bits 14:13 of the control word following the burst) are automatically calculated from the number of bytes sent. ERR has a higher priority than EOP; if EOP and ERR are both ‘1’, the EOPs for the burst is an EOP abort = ‘01’.

Table A-2: **send_user_data (PBr, SOP, EOP, Err, Addr, bytes) Inputs**

Name	Range	Description
SOP	0 or 1	Defines if the packet should begin with a SOP.
EOP	0 or 1	Defines if the packet should be terminated with an EOP.
ERR	0 or 1	Defines if the packet should be terminated with an EOP abort.
ADDR	0 to 255	Channel on which the packet should be sent.
BYTES	1 to 255	Number of bytes to send on the selected channel.

The **send_idles** procedure is used to send idle control words.

Table A-3: **send_idles (PBr, cycles) Inputs**

Name	Range	Description
CYCLES	0 to 511	Number of idle control words to send on RDat.

The **send_training** procedure is used to send training patterns.

Table A-4: **send_training (PBr, patterns) Inputs**

Name	Range	Description
PATTERNS	0 to 255	Number of training patterns to send.

The **sop_spacing** procedure is used to send errored data by sending two SOPs in less than eight cycles. This function limits the number of cycles between the two SOPs to less than seven. This ensures that a SOP spacing error occurs.

Table A-5: **sop_spacing (PBr, Bytes1, Err1, Addr1, EOP2, Err2, Addr2, Bytes2, num_cycles) Inputs**

Name	Range	Description
BYTES1	0 to 10	The number of bytes to send in the first burst. This is limited to 10 bytes to ensure SOP spacing is violated.
ERR1	0 or 1	Defines if the first packet should be terminated with an EOP abort. If set to 0 the EOPs will be calculated from BYTES1.
ADDR1	0 to 255	Channel on which the first packet should be sent.
EOP2	0 or 1	Defines if the second packet should be terminated with an EOP.
ERR2	0 or 1	Defines if the second packet should be terminated with an EOP abort. If set to 0 the EOPs will be calculated from Bytes1.

Table A-5: **sop_spacing (PBr, Bytes1, Err1, Addr1, EOP2, Err2, Addr2, Bytes2, num_cycles) Inputs *(Continued)***

Name	Range	Description
ADDR2	0 to 255	Channel on which the second packet should be sent.
BYTES2	1 to 255	The number of bytes to send in the second burst.
NUM_CYCLES	0 to [5 - roundup (BYTES1/2)]	The number of idle cycles between the first and second burst.

The **send_status** procedure is used to change the status for a particular channel.

Table A-6: **send_status (PBt, channel, value) Inputs**

Name	Range	Description
CHANNEL	0 to 255	Defines the channel for which status is updated.
VALUE	00,01,10,11	Defines the new status value to assign to the selected channel.

The **get_status** procedure is called to check status of a specific channel. It will cause the status value of that channel to be returned to the testcase.

Table A-7: **get_status (PBt, channel) Inputs**

Name	Range	Description
CHANNEL	0 to 255	Defines the channel for which status is read.

Appendix B

Verilog Details

Procedures Module

The procedures module is a package of functions instantiated in the Testcase module to simplify sending data and status to the Stimulus module. Use these functions to create any desired sequence of data or status. All functions are called from the Testcase module using the following format:

```
Format: tasks.<function name>(<inputs>)
Example: tasks.send_packet(0,40): A 40-byte long packet is sent on
channel 0.
```

The procedures module handles all clocking for the Testcase module. For an example of how these procedures are used, see the default file (`pl4_testcase.v`) provided with the core.

The tables in this section describe the supported functions included in the procedures module.

The reset procedure is used to reset the interface to the Stimulus Module. This procedure should be called at the beginning of any testcase.

The **send_packet** procedure is used to transmit an entire packet of data. This procedure will always send a SOP control word before the burst of data and an EOP control word following the data burst. The EOPS (bits 14:13 of the control word following the burst) are automatically calculated from the number of bytes sent.

Table B-1: **send_packet (Addr, bytes) Inputs**

Name	Range	Description
ADDR	0 to 255	Channel on which the packet should be sent.
BYTES	1 to 255	Number of bytes to send on the selected channel.

The `send_user_data` procedure is used to transmit a burst of data. The presence of a SOP control word (before the burst of data) and an EOP control word (following the data burst), can be specified. The EOPS (bits 14:13 of the control word following the burst) are automatically calculated from the number of bytes sent. ERR has a higher priority than EOP; if EOP and ERR are both '1', the EOPs for the burst is an EOP abort = '01.'

Table B-2: **send_user_data (SOP, EOP, Err, Addr, bytes) Inputs**

Name	Range	Description
SOP	0 or 1	Defines if the packet should begin with an SOP.
EOP	0 or 1	Defines if the packet should be terminated with an EOP.
ERR	0 or 1	Defines if the packet should be terminated with an EOP abort.
ADDR	0 to 255	Channel on which the packet should be sent.
BYTES	1 to 255	Number of bytes to send on the selected channel.

The `send_idles` procedure is used to send idle control words.

Table B-3: **send_idles (cycles) Inputs**

Name	Range	Description
CYCLES	0 to 511	Number of idle control words to send on RDat.

The `send_training` procedure is used to send training patterns.

Table B-4: **send_training (patterns) Inputs**

Name	Range	Description
PATTERNS	0 to 255	Number of training patterns to send.

The `sop_spacing` procedure is used to send erred data by sending two SOPs in less than eight cycles. This function limits the number of cycles between the two SOPs to less than seven. This ensures that a SOP spacing error occurs.

Table B-5: **sop_spacing (Bytes1, Err1, Addr1, EOP2, Err2, Addr2, Bytes2, num_cycles) Inputs**

Name	Range	Description
BYTES1	0 to 10	The number of bytes to send in the first burst. This is limited to 10 bytes to ensure SOP spacing is violated.
ERR1	0 or 1	Defines if the first packet should be terminated with an EOP abort. If set to 0 the EOPs will be calculated from BYTES1.
ADDR1	0 to 255	Channel on which the first packet should be sent.
EOP2	0 or 1	Defines if the second packet should be terminated with an EOP.

Table B-5: **sop_spacing (Bytes1, Err1, Addr1, EOP2, Err2, Addr2, Bytes2, num_cycles) Inputs *(Continued)***

Name	Range	Description
ERR2	0 or 1	Defines if the second packet should be terminated with an EOP abort. If set to 0 the EOPs will be calculated from Bytes1.
ADDR2	0 to 255	Channel on which the second packet should be sent.
BYTES2	1 to 255	The number of bytes to send in the second burst.
NUM_CYCLES	0 to [5 - roundup (BYTES1/2)]	The number of idle cycles between the first and second burst.

The `send_status` procedure is used to change the status for a particular channel.

Table B-6: **send_status (channel, value) Inputs**

Name	Range	Description
CHANNEL	0 to 255	Defines the channel whose status will be updated.
VALUE	00,01,10,11	Defines the new status value to assign to the selected channel.

The `get_status` procedure is called to check status of a specific channel. It will cause the status value of that channel to be returned to the Testcase.

Table B-7: **get_status (channel) Inputs**

Input	Range	Description
CHANNEL	0 to 255	Defines the channel whose status will be read.

Random Testcase Sample Code

The following code is an example that can be inserted into the `pl4_testcase.v` file to send randomized data to the Sink core. It should replace the default code used to send data. In addition to sending randomized data, it also randomly asserts each request signal.

```
wait (Reset_n == 1);
@ (posedge RDClk2x);

//*********************************************************************
******
// Sends out randomized data, idles, or training.
// It also randomly toggles TCIdleRequest, TCTrainingRequest,
// TCDIP4Request, TCDIP2Request, and TCSnkDip2ErrRequest
//*********************************************************************
******
forever
begin
  RandTask = {$random(`RANDOM_SEED + $time)} % 4;
  RandIdleRequest = {$random(`RANDOM_SEED + $random(`RANDOM_SEED +
$time))} % 100;
  RandTrainingRequest = {$random(`RANDOM_SEED + $time)} % 100;
  RandDIP4Request = {$random(`RANDOM_SEED + $time +
$random(`RANDOM_SEED))} % 100;
  RandDIP2Request = {$random($random(`RANDOM_SEED) + $time)} % 100;
  RandSnkDip2ErrRequest = {$random(`RANDOM_SEED + $random($time))} %
100;

  //Randomly set TCIdleRequest to 1
  if ((RandIdleRequest == 0) || (TCIdleRequest == 1))
  begin
    if (TCIdleRequest == 1)
    begin
      if (IdleRequestCnt > 0)
      begin
        IdleRequestCnt <= IdleRequestCnt - 1'b1;
        TCIdleRequest <= 1'b1;
      end
      else
      begin
        IdleRequestCnt <= 'b0;
        TCIdleRequest <= 1'b0;
      end
    end
    else
    begin
      TCIdleRequest <= 1'b1;
      IdleRequestCnt <= {$random(`RANDOM_SEED + $time)} % 9;
    end
  end

  //Randomly set TCTrainingRequest to 1
  if ((RandTrainingRequest == 0) || (TCTrainingRequest == 1))
  begin
    if (TCTrainingRequest == 1)
    begin
      if (TrainingRequestCnt > 0)
      begin
        TrainingRequestCnt <= TrainingRequestCnt - 1'b1;
        TCTrainingRequest <= 1'b1;
      end
      else
      begin
        TrainingRequestCnt <= 'b0;
        TCTrainingRequest <= 1'b0;
      end
    end
    else
    begin
      TCTrainingRequest <= 1'b1;
      TrainingRequestCnt <= {$random(`RANDOM_SEED + $time)} % 9;
    end
  end

  //Randomly set TCDIP4Request to 1
  if ((RandDIP4Request == 0) || (TCDIP4Request == 1))
  begin
    if (TCDIP4Request == 1)
  begin
    if (DIP4RequestCnt > 0)
    begin
      DIP4RequestCnt <= DIP4RequestCnt - 1'b1;
      TCDIP4Request <= 1'b1;
    end
    else
    begin
      DIP4RequestCnt <= 'b0;
      TCDIP4Request <= 1'b0;
    end
  end
  else
  begin
    TCDIP4Request <= 1'b1;
    DIP4RequestCnt <= {$random(`RANDOM_SEED + $time)} % 9;
  end
end

//Randomly set TCDIP2Request to 1
if ((RandDIP2Request == 0) || (TCDIP2Request == 1))
begin
  if (TCDIP2Request == 1)
  begin
    if (DIP2RequestCnt > 0)
    begin
      DIP2RequestCnt <= DIP2RequestCnt - 1'b1;
      TCDIP2Request <= 1'b1;
    end
    else
    begin
      DIP2RequestCnt <= 'b0;
      TCDIP2Request <= 1'b0;
    end
  end
  else
  begin
    TCDIP2Request <= 1'b1;
    DIP2RequestCnt <= {$random(`RANDOM_SEED + $time)} % 9;
  end
end

//Randomly set TCSnkDip2ErrRequest to 1
if ((RandSnkDip2ErrRequest == 0) || (TCSnkDip2ErrRequest == 1))
begin
  if (TCSnkDip2ErrRequest == 1)
  begin
    if (SnkDip2ErrRequestCnt > 0)
    begin
      SnkDip2ErrRequestCnt <= SnkDip2ErrRequestCnt - 1'b1;
      TCSnkDip2ErrRequest <= 1'b1;
    end
    else
    begin
      SnkDip2ErrRequestCnt <= 'b0;
      TCSnkDip2ErrRequest <= 1'b0;
    end
  end
  else
```

```
    begin
      TCSnkDip2ErrRequest <= 1'b1;
      SnkDip2ErrRequestCnt <= {$random(`RANDOM_SEED + $time)} % 9;
    end
  end

  //Sends a random sized packet to a random channel
  if (RandTask == 0)
  begin
    tasks.send_packet({$random(`RANDOM_SEED + $time)} % (`NUM_CHANNELS
- 1), ($random(`RANDOM_SEED + $time) % 255) + 1'b1);
  end
  //Sends a random sized packet to a random channel. Also SOP, EOP, and
  //Err are randomized
  else if (RandTask == 1)
  begin
    tasks.send_user_data({$random(`RANDOM_SEED + $time)} % 2,
{$random(`RANDOM_SEED + $time + $random(`RANDOM_SEED))} % 2,
{$random(`RANDOM_SEED + $time + $random(`RANDOM_SEED + $time))} % 2,
{$random(`RANDOM_SEED + $time)} % (`NUM_CHANNELS - 1),
($random(`RANDOM_SEED + $time) % 255) + 1'b1);
  end
  //Sends a random number of idles to the Sink Core
  else if (RandTask == 2)
  begin
    tasks.send_idles(({$random(`RANDOM_SEED + $time)} % 10) + 1);
  end
  //Sends a random number of training patterns to the sink core
  else if (RandTask == 3)
  begin
    tasks.send_training(({$random(`RANDOM_SEED + $time)} % 10) + 1);
  end
  else
  begin
    @ (posedge RDClk2x);
    $display("Out of Range: %0d", $time);
  end
end
```

Appendix C

Data and Status Monitor Warnings

The Data and Status monitors continuously check data sent to and received from the demonstration test bench. There are several common warnings that occur when the Testcase module is modified. The warnings are listed and described below.

TDat Warning: Source is segmenting packets <*simulation time*>

This warning means that the Source core is sending payload resumes in the middle of sending a burst. This is acceptable operation if `SrcBurstMode` = 0. If `SrcBurstMode` = 1, this should only occur if the maximum burst length is reached (as defined by `SrcBurstLen`).

RStat Info: Sink is out of frame. Expect TDat mismatches <*simulation time*>

This indicates that the Sink core went out of frame during operation. Unless training or idles are being sent on `RDat` when this occurs, there will be data errors on TDat. This is because what is being sent in on `RDat` is no longer being transferred to TDat.

RStat Info: Expected DIP2 mismatch received: `SnkDip2ErrReqFlag` = 1 <*simulation time*>

This indicates that a `DIP2` error was detected on `RStat`. It is only a note and not an error because `SnkDip2ErrReq` was asserted, which means that a `DIP2` error is expected.

RDat Warning: Protocol Violation #4. Idle follows data on a non-credit boundary <*simulation time*>

This indicates that the SPI-4.2 protocol was violated when data was sent from the demonstration test bench. The most likely cause is that `send_user_data` was used to send data without an EOPS, which ended on a non-credit boundary, then an idle was sent using `send_idles`.

RDat Warning: Protocol Violation

Any RDat protocol violation occurred because of incorrectly formatted data transmitted from the Testcase Module (that is, they are user-created).

Appendix D

Timing Simulation Warning and Error Messages

There are several common simulation warnings and error messages when timing simulation is run on the example design. These warnings and messages are described in this appendix.

"# TDat Error: Data Mismatch # 4. Expected 000f, Received 000x. 339280 ps"

The data mismatch results from the data going to unknown "x" state. To prevent "x" from propagating in your simulation, use the "+no_notifier" option to `vsim` command when using ModelSim Simulator (MTI). If you are using other simulators, consult the manufacturer documentation for possible ways to turn off "x" propagation.

SETUP, HOLD, RECOVERY violation on /X_FF

These violations might come from either the Sink core or Source Core, and they originated from register elements that are transiting between two clock domains. These timing violations can be safely ignored.

When running simulation on a SPI-4.2 Sink Core with Global Clocking and DPA Clock Adjustment option, the signal `Locked_RDClk` (from RDClk DCM) might get deasserted after `PhaseAlignRequest` is asserted. When the `PhaseAlignRequest` has been asserted, the IDELAY goes through the reset process and the clock stops toggling momentarily. This might cause the lock signal from the DCM to get deasserted in simulation (this does not occur in hardware testing). `Locked_RDClk` should be ignored after the `PhaseAlignRequest` has been asserted in simulation.

"Memory Collision Error on X_RAMB16"

The "Memory Collision" error occurs occasionally because the calendar block is trying to read out values at the same time that you are writing them in; however, this is not a problem because you are only supposed to write the calendar when the core is disabled.